The MDELab Tool Framework for the Development of Correct Model Transformations with Triple Graph Grammars*

Stephan Hildebrandt, Leen Lambers, and Holger Giese
Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
[stephan.hildebrandt|leen.lambers|holger.giese]@hpi.uni-potsdam.de

1. INTRODUCTION

Model transformations play one of the key roles in Model-Driven Engineering (MDE). Therefore, their correctness is of major importance. Triple Graph Grammars[6] (TGGs) are an important representative of a relational model transformation specification technique for describing bidirectional transformations. A TGG specifies relations between source and target models, but cannot be executed directly to perform a corresponding model transformation. Instead, a *TGG Implementation* has to be derived, which must be *conform* to the TGG, i.e., the target model it produces with derived operational rules for a given source model must also be a valid target model for that source model according to the TGG. TGG implementations can be generated from the TGG specification for performing forward and backward model transformations, but also to perform model integration as well as synchronization.

In this paper, we present a tool framework (cf. Figure 1) that we developed as part of our *MDELab*[1] tool set for developing correct TGG-based model transformations. Thereby, we aim at *verifying correctness properties* on the specification level (cf.Section 2), i.e. for a particular TGG, and ensure that these correctness properties can be carried over to the implementation by *conformance checking* (cf.Section 3). As a consequence, correctness is shown independently of the input model of a model transformation. Moreover, we support the *developers* of the TGG specification and TGG implementation to find errors already at design time, so that the transformation *user* is spared from errors at runtime as much as possible.

*This work was developed in the course of the project - Correct Model Transformations - Hasso Plattner Institut, Universität Potsdam and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft. See `http://www.hpi.uni-potsdam.de/giese/projekte/kormoran.html?L=1`.

[1] `http://www.mdelab.de`

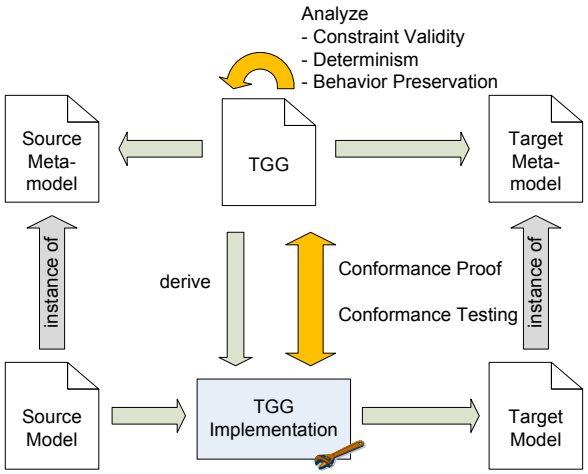

Figure 1: Overview

2. ANALYSIS

We concentrate on analyzing TGGs for three key properties of model transformations in MDE: determinism, validity with meta-model constraints, and behavior preservation.

2.1 Determinism

A forward (backward) deterministic TGG relates a unique target model to a given source model (and the other way round). In practice, model transformations are mostly required to be a function. Therefore, knowing whether a TGG is deterministic is an important information. In addition, a TGG implementation of a deterministic TGG does not need backtracking, which eases developing the implementation and increases performance. Termination of model transformations derived from a TGG is guaranteed in our framework by checking simple syntactic criteria on the TGG rules [1]. To check result uniqueness, we can export a TGG to AGG[2], a graph transformation and analysis tool. AGG checks the derived TGG operational rules for conflicts, i.e., situations, where a model element can be transformed by more than one rule, potentially leading to non-determinism.

2.2 Validity of TGGs with Constraints

Metamodels often contain OCL constraints, which express additional conditions that models have to fulfill to be valid

[2] http://user.cs.tu-berlin.de/ gragra/agg/

instances of that metamodel. In general, a model transformation is expected to always output a valid target model for a valid source model (*forward validity*). To check forward validity of a TGG, we provide an *Invariant Checker* and a *Counter Example Generator* [3]. Both tools search for pairs of corresponding source and target models, where the source model is valid but the target is not. The *Invariant Checker* performs a complete static analysis, but cannot handle arbitrary OCL expressions. Such expressions have to be translated to graph patterns, which is only possible for a subset of OCL, due to the higher expressiveness of OCL. The *Counter Example Generator* generates counter example models by randomly applying TGG rules and validating all metamodel constraints on these models afterwards. This allows to handle arbitrary OCL expressions, but, of course, this is an incomplete analysis because the number of models that can be generated by a TGG is, in general, infinite.

2.3 Behavior Preservation

When transforming behavior models, the output model of a model transformation is often expected to behave like the input model, i.e., the transformation preserves the model's behavior. Usually, automated behavior preservation verification techniques either show that specific properties are preserved, or more generally and complex, they show some kind of bisimulation between source and target model of the transformation. Both kinds of behavior preservation verification goals have been presented with automatic tool support for the instance level. However, we aim at developing automatic verification approaches for the transformation level, i.e., for all source and target models specified by the model transformation. In [2] we presented a first approach toward automatic behavior preservation verification on the transformation level for model transformations specified by TGGs and semantic definitions given by graph transformation rules. In particular, we show that the behavior preservation problem can be reduced to invariant checking for graph transformation. Investigating the limitations of this approach and integrating this analysis technique into our tool framework is part of current work.

3. CONFORMANCE

We have developed a proof as well as an automatic testing framework for checking conformance of the TGG with its implementation.

3.1 Conformance Proof

A TGG implementation often introduces several optimizations to improve transformation performance, e.g., bookkeeping mechanisms. For our own TGG implementation, we have proven[1] the conformance of the implemented optimizations with the semantics of TGGs. We have formulated several conditions, that a TGG must fulfill so that the derived implementation is conform. Most of these conditions imply simple syntactical checks on the TGG rules. However, it is also required that the TGG rules imply deterministic model transformations. This involves a more complex check using critical pair analysis in AGG as mentioned in Subsection 2.1. In case of scalability problems, we also developed a run-time check [1] verifying for a particular model transformation instance the uniqueness of its result.

3.2 Conformance Testing

Occasionally, there is no formal conformance proof available for a TGG implementation. But even if so, it may be uncertain whether each formal concept is realized correctly by the implementation. Moreover, usually, available formal concepts neither cover every technicality, nor cover each additional optimization an implementation relies on. Therefore, we have proposed an automatic conformance testing framework[4] for TGG implementations and have implemented it for our own TGG implementation. The whole process of generating the test input models and test oracles, executing the transformation, and comparing the output model with the oracle is completely automated. In addition, the framework can also output specification and implementation coverage data. This data helps the user assess the quality of a set of test cases.

4. CONCLUSION

In this tool paper we characterize our framework to foster the development of correct model transformations. In this framework we exploit the sound formal foundation of TGGs[1, 6] to support advanced analysis capabilities to ensure determinism, the generation of valid results, and behavior preservation. These results, based on the formal model of TGGs, are linked with the implementation level by a conformance proof for our TGG implementation and a conformance testing approach that is also applicable to other TGG implementations. As future work it is planned to further improve the expressiveness of TGGs, tool integration, and attribute handling (cf. [5]).

Acknowledgements

We would like to thank Thomas Beyhl, Basil Becker, and Johannes Dyck and all other students involved in the project.

5. REFERENCES

[1] H. Giese, S. Hildebrandt, and L. Lambers. Bridging the gap between formal semantics and implementation of triple graph grammars. *Software and Systems Modeling, Springer Berlin / Heidelberg*, pages 1–27, 2012.

[2] H. Giese and L. Lambers. Towards automatic verification of behavior preservation for model transformation via invariant checking. In *Proc. of ICGT 2012*, LNCS. Springer, 2012. to appear.

[3] S. Hildebrandt, L. Lambers, B. Becker, and H. Giese. Integration of triple graph grammars and constraints. In *Proc. of GraBaTs 2012*, 2012. To appear.

[4] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick, and I. Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In *Proc. of AGTIVE 2011*, volume 7233. Springer, 2012. To appear.

[5] L. Lambers, S. Hildebrandt, H. Giese, and F. Orejas. Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case. In *Proc. of 1st International Workshop on Bidirectional Transformations*. EC-EASST, 2012. To appear.

[6] A. Schürr. Specification of graph translators with triple graph grammars. In *Proc. of* 20^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Spinger, 1994.